10_Fine-Tuning

May 6, 2025

Computación Inteligente:

Grandes Modelos de Lenguajes

Clase 10: Técnicas de Fine-Tuning Eficiente (LoRA y Adapters)

0.1 1. Introducción

1. **El Poder de los Modelos Pre-entrenados**
   - Revisar brevemente el concepto de modelos pre-entrenados a gran escala (ej. BERT, GPT, LLaMA, T5, etc.).
   - Son la base de muchos avances recientes en PNL, Visión por Computadora, etc.
   - Han aprendido representaciones ricas y generales de los datos.
2. **El Desafío de la Adaptación a Tareas Específicas**
   - Un modelo pre-entrenado no es *directamente* óptimo para una tarea *particular* o un dominio de datos *específico* sin adaptación.
   - Necesitamos "enseñarle" al modelo cómo aplicar sus conocimientos generales a nuestra tarea (clasificación, traducción, generación de texto específico, etc.).
3. **La Solución Tradicional: Full Fine-Tuning**
   - Congelar las capas de embedding o las primeras capas y entrenar el resto del modelo en los nuevos datos.
   - O, más comúnmente, descongelar *todas* las capas y entrenar *todo* el modelo con una tasa de aprendizaje baja en los nuevos datos.
4. **Problemas del Full Fine-Tuning**
   - **Costo Computacional:** Entrenar miles de millones de parámetros requiere GPUs potentes y mucho tiempo.
   - **Memoria:** Cargar un modelo completo y sus optimizadores (con estados como el momentum) consume una gran cantidad de memoria VRAM.
   - **Almacenamiento:** Cada tarea requiere una copia *completa* del modelo ajustado. Esto escala muy mal si tenemos muchas tareas.
   - **Catastrophic Forgetting (Olvido Catastrófico):** Ajustar todas las capas puede hacer que el modelo "olvide" habilidades generales aprendidas durante el pre-entrenamiento, especialmente si los datos de la nueva tarea son muy diferentes o

limitados.

- **Dificultad para compartir:** Es difícil compartir o combinar adaptaciones hechas a diferentes copias del modelo base.

5. **Introduciendo las Técnicas de Adaptación Eficiente de Parámetros (PEFT - Parameter-Efficient Fine-Tuning**
   - El objetivo es adaptar un modelo pre-entrenado a una nueva tarea o dominio entrenando *muy pocos* parámetros adicionales o modificando *muy pocas* capas existentes.
   - Mantener la mayor parte del modelo pre-entrenado *congelado*.
   - Resultados a menudo competitivos con el full fine-tuning, pero con una fracción del costo
   - Nos centraremos en dos técnicas PEFT prominentes y representativas: LoRA (Low-Rank Adaptation) y Adapters.

0.2 2. Fundamentos de PEFT

Idea Central: En lugar de modificar todo el espacio de parámetros W de una capa, buscamos aprender una *pequeña* modificación ΔW o añadir un *pequeño* módulo M.
- **Categorías Generales de PEFT (Breve):**
- **Adición de Nuevos Parámetros/Módulos:** Se insertan pequeñas redes o módulos dentro del modelo existente (ej. Adapters, prompts/prefixes entrenables).
- **Reparameterización:** Se redefine cómo se aplican las actualizaciones a los pesos existentes, generalmente de una manera de bajo rango (ej. LoRA).
- **Selección de Capas/Parámetros:** Entrenar solo un subconjunto específico de capas o parámetros del modelo original (menos común ahora para LLMs muy grandes debido a la interacción entre capas).
- **Beneficios Clave de PEFT:**
- **Reducción Drástica de Parámetros Entrenables:** Se entrena un 0.1% a 5% de los parámetros totales, en lugar del 100%.
- **Menor Consumo de Memoria (VRAM):** Solo necesitas calcular gradientes y estados del optimizador para los pocos parámetros entrenables.
- **Menor Requerimiento de Almacenamiento:** Guardar solo los parámetros entrenables (o la matriz de bajo rango) por tarea.
- **Entrenamiento Más Rápido:** Menos parámetros -> menos cálculos de gradiente -> entrenamiento más rápido.
- **Mitigación del Olvido Catastrófico:** Al mantener el modelo base congelado, se preservan las habilidades generales.
- **Composabilidad (potencial):** Algunas técnicas permiten "cambiar" rápidamente la adaptación para una tarea diferente.

0.3 3. LoRA (Low-Rank Adaptation)

- **La Intuición:** Investigaciones sugieren que las actualizaciones a los pesos de una red neuronal durante el fine-tuning tienen intrínsecamente una *dimensión intrínseca baja* o *rango bajo*. Es decir, la información relevante para la adaptación puede ser capturada en un subespacio de menor dimensión.

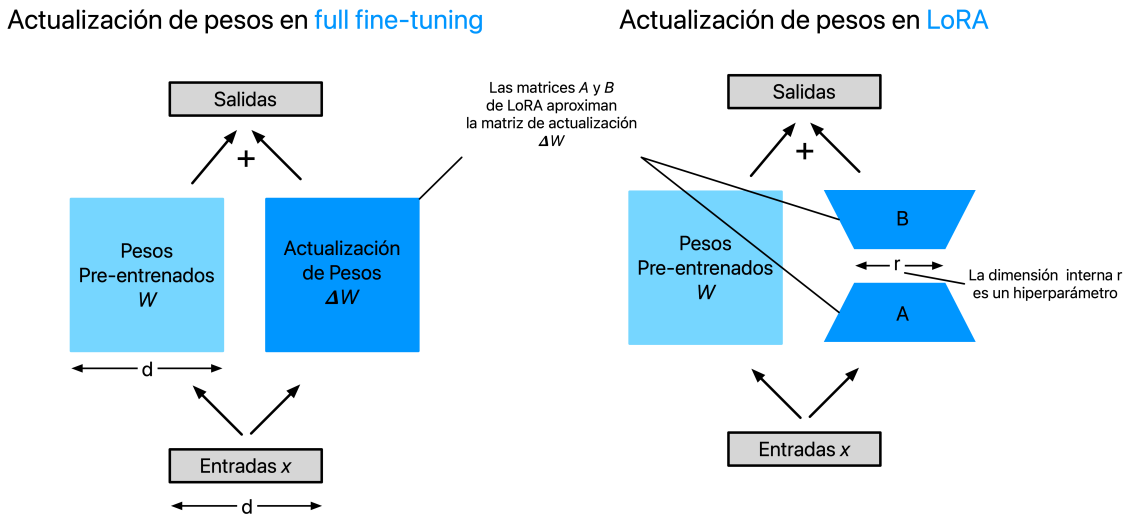
1	2	5	3	4
3	3	9	6	9
2	3	8	5	7
4	1	6	5	9

=

1	2
3	3
2	3
4	1

1	0	1	1	2
0	1	2	1	1

- **La Idea de LoRA:** En lugar de aprender directamente el cambio completo de la matriz de pesos $\Delta W \in R^{d \times k}$, se descompone esta actualización en el producto de dos matrices más pequeñas de bajo rango: $\Delta W = AB$.
* $A \in R^{d \times r}$
* $B \in R^{r \times k}$
* Donde r es el "rango" de la adaptación, y $r \ll min(d, k)$.
* **Cómo Funciona en la Práctica:**
* Consideremos una matriz de pesos $W_0 \in R^{d \times k}$ en una capa (ej. la matriz de pesos de la capa de atención 'query' o 'value').
* Durante el fine-tuning, queremos actualizarla a $W_0 + \Delta W$.
* Con LoRA, mantenemos W_0 congelada y aprendemos las matrices A y B.
* La operación hacia adelante de la capa se convierte en $h = W_0\ x + \Delta W x = W_0\ x + ABx$.
* x es la entrada a la capa. h es la salida. * **Arquitectura/Integración:**
* Las matrices A y B se *añaden* a la capa original, pero se implementan de forma que el cálculo sea eficiente.
* Típicamente, se aplica a las matrices de proyección en las capas de auto-atención (Query, Key, Value, Output proyections) y, a veces, a las capas feed-forward.
* La entrada x primero pasa por W_0. Simultáneamente, x pasa por B, luego por A, y el resultado se *suma* a la salida de W_0.
* Visualización: Diagrama mostrando la bifurcación: $x \to W_0 \to output$ y $x \to B \to A \to output$, con las dos ramas sumándose.



- Es común inicializar A con valores aleatorios pequeños y B con ceros, para que la adaptación inicial $\Delta W = AB$ sea cero y no perturbe al modelo pre-entrenado al principio.

- **Entrenamiento con LoRA:**
  - Solo se entrenan los parámetros de las matrices A y B (y opcionalmente los parámetros de LayerNorm o bias si se consideran parte de la adaptación).
  - Los parámetros de W_0 se mantienen *congelados*.
  - La retropropagación solo calcula gradientes para A y B.
  - El número de parámetros entrenables es $d \times r + r \times k$. Esto es mucho menor que $d \times k$ si r es pequeño.
- **Inferencia con LoRA:**
  - Para una inferencia más rápida, la actualización $\Delta W = AB$ puede ser *fusionada* (merged) en la matriz W_0 : $W_{final} = W_0 + AB$.
  - Una vez fusionada, la operación es idéntica a la de la capa original $W_{final}\ x$, sin latencia adicional.
  - La ventaja de la fusión es la velocidad. La desventaja es que pierdes la flexibilidad de cambiar rápidamente de adaptación si tienes múltiples tareas.
  - Para cambiar de tarea sin reiniciar el modelo completo, puedes cargar un conjunto diferente de matrices A y B. Esto se conoce como "adapter swapping" en el contexto de LoRA.
- **Ventajas de LoRA:**
  - Muy pocos parámetros entrenables (alta eficiencia de memoria y almacenamiento).
  - Entrenamiento rápido.
  - Rendimiento a menudo comparable al full fine-tuning.
  - Posibilidad de fusionar pesos para inferencia sin latencia adicional (después de la fusión).
  - Fácil composabilidad/intercambiabilidad de adaptaciones para diferentes tareas (si no se fusionan).
  - Especialmente efectivo para tareas generativas.
- **Desventajas de LoRA:**
  - Introduce una pequeña latencia *durante el entrenamiento* o si no se fusionan las matrices para inferencia.
  - El "rango" r es un hiperparámetro crucial que requiere ajuste. Un r muy bajo puede no capturar suficiente información, un r muy alto reduce la eficiencia.
  - Principalmente aplicable a capas que involucran multiplicaciones de matrices (lineales, convolucionales, pero más común en atención).

0.4 4. Adapters

- **La Intuición:** Insertar pequeñas redes neuronales ("módulos adapter") dentro de un modelo pre-entrenado, entrenando solo estos módulos.

- **La Idea de Adapters:** Añadir pequeñas capas feed-forward con una conexión residual *después* de las subcapas principales del modelo (ej. después de la auto-atención o después de la capa feed-forward principal en un Transformer).
- **Cómo Funciona en la Práctica (Arquitectura Clásica - Serial):**
  - Considera una subcapa f (ej. auto-atención) seguida de una normalización de capa y conexión residual en un Transformer. La salida es $y = LayerNorm(x + f(x))$.

  - Con Adapters, se inserta un módulo adicional *dentro* de la rama residual, típicamente después de la primera activación o justo antes de la adición final: $y = LayerNorm(x + f(x) + Adapter(f(x)))$ o $y = LayerNorm(x + Adapter(f(x) + x))$.

  - La estructura del módulo Adapter es típicamente un "cuello de botella" (bottleneck):
    * Una capa lineal de proyección hacia abajo (down-projection) que reduce la dimensión del vector de entrada ($d \to m$, donde $m \ll d$).

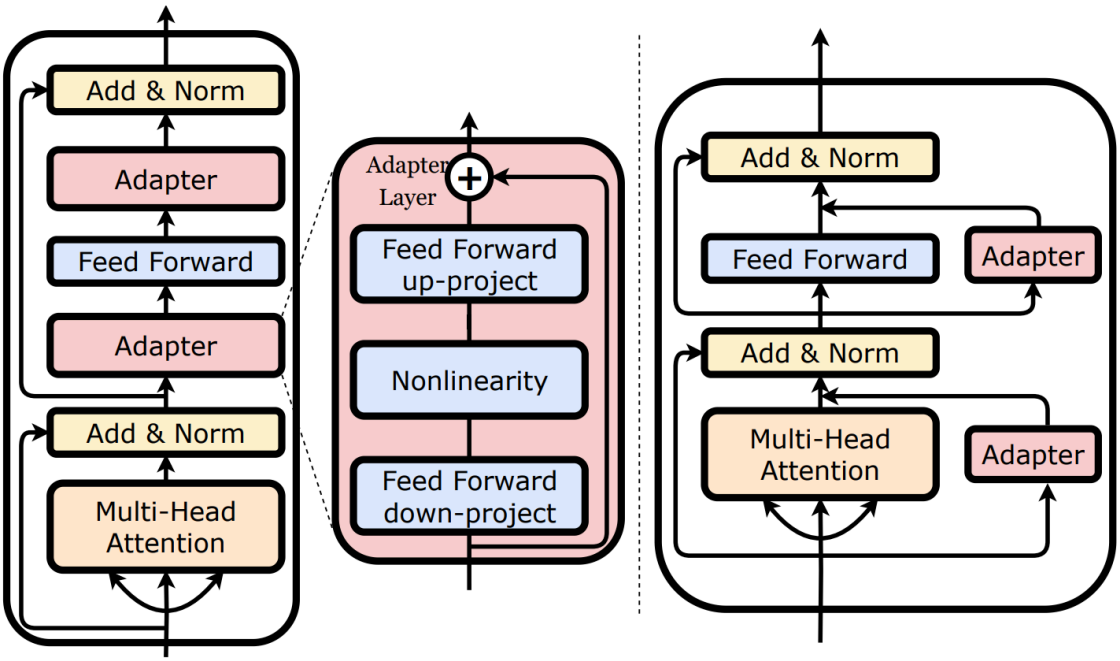
    * Una función de activación no lineal (ej. GELU, Swish).

    * Una capa lineal de proyección hacia arriba (up-projection) que restaura la dimensión original ($m \to d$).

    * A menudo, una conexión residual *dentro* del propio módulo adapter para mejorar la estabilidad del entrenamiento.

    * Una normalización de capa (LayerNorm) antes o después del módulo adapter.

  - Visualización: Diagrama mostrando la inserción del módulo adapter después de una subcapa, con la estructura interna del bottleneck.



- **Tipos de Arquitecturas de Adapter (Breve):**
  - **Serial Adapters:** Insertados secuencialmente en la rama residual. (El tipo clásico).
  - **Parallel Adapters:** Colocados en paralelo con la subcapa original, sumando sus salidas. Similar conceptualmente a LoRA en la suma, pero la implementación es diferente (red pequeña vs. descomposición de matriz).
  - Otras variaciones existen (ej. AdapterFusion para combinar adapters de múltiples tareas).
- **Entrenamiento con Adapters:**
  - Solo se entrenan los parámetros de los módulos Adapter (las matrices de pesos de las proyecciones hacia abajo y hacia arriba, y los parámetros de LayerNorm dentro o alrededor del adapter).
  - Los parámetros del modelo base pre-entrenado se mantienen *congelados*.
  - La retropropagación solo calcula gradientes para los parámetros del adapter.
  - El número de parámetros entrenables depende del número de adapters insertados y del tamaño del cuello de botella (m). Típicamente, dos adapters por bloque Transformer (uno después de atención, uno después de FFN). El número de parámetros de un adapter es $d \times m + m \times d = 2dm$. Total es el número de adapters $\times 2dm$. Sigue siendo mucho menor que el modelo completo.
- **Inferencia con Adapters:**
  - Los módulos Adapter están *activos* durante la inferencia.
  - Esto significa que introducen latencia adicional porque se realizan cálculos adicionales (las operaciones del bottleneck).

  - No hay una forma obvia de "fusionar" la funcionalidad del adapter en los pesos del modelo base de la misma manera que LoRA, porque es una red separada con activaciones no lineales y conexiones residuales internas.
- **Ventajas de Adapters:**
  - Buena eficiencia de parámetros y almacenamiento.
  - Entrenamiento rápido.
  - Clara modularidad: Cada adapter es un componente independiente que puedes añadir o quitar.
  - Potencialmente bueno para tareas donde necesitas una adaptación más "localizada" dentro de la red.
  - Excelente para composabilidad: Puedes tener múltiples adapters y activarlos selectivamente para diferentes tareas o incluso combinarlos (AdapterFusion).
- **Desventajas de Adapters:**
  - Introduce latencia *siempre* durante la inferencia, ya que los módulos están activos.
  - Puede requerir un número ligeramente mayor de parámetros entrenables que LoRA para un rendimiento comparable, dependiendo de la configuración del cuello de botella.
  - El tamaño del cuello de botella (m) es un hiperparámetro crucial.

0.5 5. Comparación LoRA vs. Adapters

Característica	LoRA (Low-Rank Adaptation)	Adapters
Enfoque Principal	Reparameterización (aprendizaje de ΔW)	Adición de Módulos Neuronales (Bottlenecks)
Parámetros Entrenables	Muy pocos (d×r+r×k)	Pocos (2dm por módulo × #módulos)
Memoria (VRAM)	Muy eficiente	Muy eficiente
Almacenamiento	Muy eficiente (guardas A y B)	Muy eficiente (guardas pesos de Adapters)
Velocidad Entrenamiento	Rápido	Rápido
Velocidad Inferencia	Puede fusionarse en W0 (sin latencia)	Siempre introduce latencia (módulos activos)
Composabilidad	Alta (cambiar A/B o fusionar)	Muy Alta (añadir/quitar módulos, fusionarlos)
Implementación	Modifica la operación de multiplicación matriz	Inserta sub-redes con conexiones residuales
Dónde se Aplica	Principalmente capas lineales/atención	Después de sub-capas (atención, FFN)
Hiperparámetros Clave	Rango r, Tasa de aprendizaje	Tamaño cuello de botella m, Tasa de aprendizaje

- **Discusión de Trade-offs:**
  - Si la *latencia de inferencia* es crítica (aplicaciones en tiempo real), LoRA con fusión de pesos es a menudo preferible.
  - Si la *modularidad extrema* y la *combinación de tareas* son primordiales, los Adapters pueden ser más flexibles.
  - Ambos son significativamente mejores que el full fine-tuning en eficiencia.
  - La elección a menudo depende de la tarea específica, el modelo base y las restricciones de hardware/implementación. En la práctica, ambos pueden dar resultados similares en rendimiento de la tarea si se ajustan bien los hiperparámetros.

0.6 6. Otras Técnicas PEFT

- **Prompt/Prefix Tuning:** No modifica los pesos del modelo base en absoluto. Aprende un pequeño conjunto de vectores continuos ("prompts" o "prefixes") que se concatenan a las entradas del modelo (embeddings o claves/valores de atención). Son muy eficientes en parámetros, pero a veces no alcanzan el rendimiento de LoRA o Adapters, especialmente en tareas de generación.
- **IA³ (Infused Adapter by Inhibiting and Amplifying Activations):** Escala activaciones (claves, valores, FFN) con vectores entrenables, en lugar de añadir o modificar matrices de pesos directamente. También muy eficiente.
- **QLoRA:** Una optimización de LoRA que usa cuantización (ej. 4-bit) del modelo base congelado para reducir aún más el uso de memoria, mientras aplica LoRA a los pesos cuantizados. Muy popular actualmente para fine-tuning en hardware limitado.

0.7 7. Consideraciones Prácticas e Implementación

- **Seleccionando la Técnica:** ¿Cuándo usar LoRA? ¿Cuándo Adapters? ¿Cuándo QLoRA? Discutir los criterios (latencia, modularidad, memoria disponible, rendimiento esperado).
- **Hiperparámetros:** La importancia de ajustar el rango r (LoRA) o el tamaño del cuello de botella m (Adapters), la tasa de aprendizaje, el número de épocas, etc.
- **¿Dónde aplicar LoRA/Adapters?:** En Transformers, es común aplicarlos en las capas de atención (Q, K, V, O) y a veces en las capas FFN. La investigación sigue explorando las mejores ubicaciones.
- **Herramientas y Bibliotecas:**
  - **Hugging Face `peft` library:** Es la libreria *de facto* para implementar fácilmente muchas técnicas PEFT, incluyendo LoRA y Adapters, sobre modelos de la librería transformers. Demostrar o mencionar cómo instanciar un modelo base y envolverlo con una configuración PEFT (ej. LoraConfig).
  - Integración con librerías de entrenamiento como transformers.Trainer o PyTorch

Lightning.

- **Hardware:** Aunque PEFT reduce drásticamente los requisitos, sigue siendo útil tener una GPU para entrenar incluso los pocos parámetros. La ventaja es que GPUs más pequeñas o en menor cantidad pueden ser suficientes.

0.8 8. Tendencias Actuales y Futuro

- Combinación de técnicas PEFT (ej. LoRA + Prompt Tuning).
- Aplicación de PEFT a modelos multimodales.
- Investigación en nuevas arquitecturas PEFT aún más eficientes o con mejor rendimiento.
- La importancia creciente de PEFT para la democratización del acceso al fine-tuning de modelos muy grandes.

0.9 9. Conclusión

- Recapitulando: El full fine-tuning es costoso y a menudo innecesario para adaptar modelos grandes.
- PEFT ofrece alternativas eficientes: LoRA (reparameterización de bajo rango) y Adapters (módulos insertados).
- Ambos tienen sus fortalezas y debilidades (latencia vs. modularidad).
- Son herramientas esenciales para trabajar con modelos de lenguaje grandes y otros modelos fundacionales en la práctica.

[ ]: